

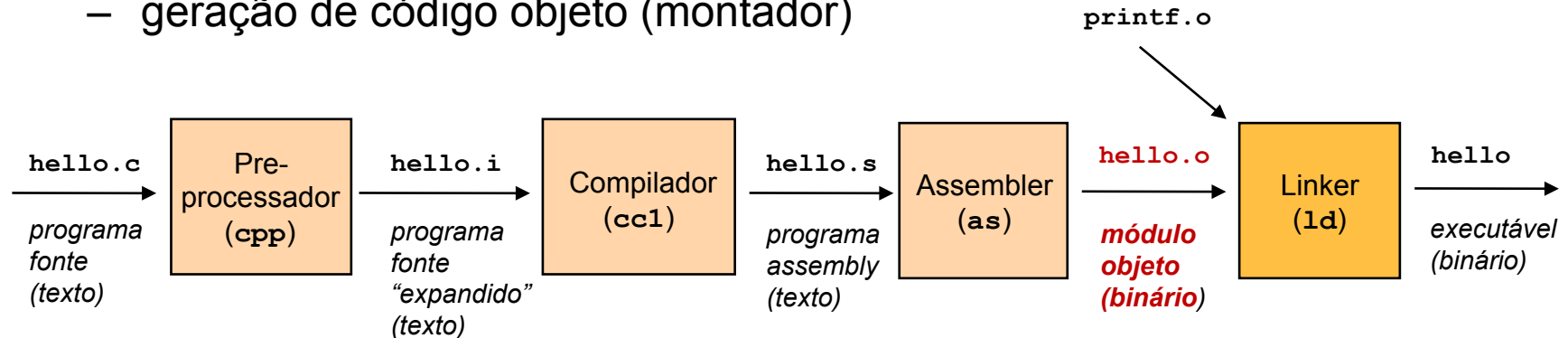
Compilação, Amarração, Relocação e Carga

Desenvolvimento em módulos

- Cada módulo contém uma coleção de tipos, constantes, **variáveis** e **procedimentos**
 - compilado em separado
 - deve se “articular” com os demais, na formação de um executável
 - módulos exportam e importam “objetos” (variáveis e procedimentos)
- O compilador gera código objeto para cada módulo, e o processo de **amarração** (*linkedição*) une os objetos em um executável
 - é esse processo que permite a **compilação em separado** !

Compilação e Amarração

- A compilação de um módulo C compreende:
 - pré-processamento
 - geração de código em *assembly* (compilador)
 - geração de código objeto (montador)



- Um módulo objeto não pode ser executado:
 - dependência de outros módulos e/ou bibliotecas (dados, código): as lacunas são "preenchidas" pela **amarração**
 - os endereços não são definitivos: referências são "corrigidas" pela **relocação**
 - compilador/montador gera código /dados com todas as seções em 0

Contéudo de um Arquivo Objeto

- Além de “seções” de código e dados, um arquivo objeto contém informações que permitem a “amarração” com outros módulos
 - **símbolos exportados**
 - funções e variáveis **definidas** pelo módulo
 - **referências externas**
 - funções e variáveis **referenciadas** pelo módulo
 - **dicionário de relocação**
 - entradas para cada posição (*offset*) de instruções e dados que referenciam a memória
- Pode conter, opcionalmente, informação para *debugger*
 - variáveis locais e globais, tipos das variáveis, código fonte, ...
 - mapeamento programa fonte (linhas) X instruções máquina

Formato ELF (relocável)

- Formato padrão para diversos “variantes” de Unix (*executable and linkable*)
 - conceitos básicos comuns a outros formatos
- Cabeçalho (*header*) *ELF*
 - tipo de arquivo (.o, exec), arquitetura (ex. IA32), localização da tabela de seções
- Tabela de seções
 - localização e tamanho das seções
- **symtab** : tabela de símbolos
 - funções e variáveis exportadas e externas
- **rel.text** e **rel.data**
 - informação de relocação: instruções e variáveis (inicializadas) que devem ser modificadas

ELF header
.text
.rodata
.data
.bss
.symtab
.rel.text
.rel.data
.debug e .line
.strtab
Section header table

Amarração (*linking*)

- Combina vários módulos objeto (.o) em um único arquivo executável
- Tarefas principais:
 - resolução das referências externas dos módulos
 - união dos módulos em um arquivo executável
 - relocação das referências absolutas à memória
- Módulos de bibliotecas estáticas também são incluídos
 - uma biblioteca estática reúne módulos objeto relacionados (ex. libc.a, libm.a)
 - para criar uma biblioteca, basta listar os arquivos objeto, por exemplo:

```
ar rcs mylib.a addvector.o multvector.o
```

Resolução de referências

- Cada referência externa deve ser associada a uma (única) exportação de símbolo **de mesmo nome** especificada em algum módulo
 - as tabelas de símbolos de cada módulo contém
 - símbolos exportados: nome, tamanho, dado/função, seção, *offset*,...
 - referências externas: **nome**
- Se nenhum símbolo exportado com esse nome é encontrado, o *linker* termina com erro
- Se há duas exportações com o mesmo nome, o *linker* também termina com erro

Exemplos

```
void foo(void);  
  
int main() {  
    foo();  
    return 0;  
}
```



```
/tmp/cc0cCWXd.o: In function `main':  
teste.c:(.text+0x7): undefined reference to `foo'  
collect2: ld returned 1 exit status
```

```
int i = 1;  
  
int foo(int j) {  
    return i + j;  
}
```



```
int foo(int);  
int i = 0;  
  
int main() {  
    int k = foo(i);  
    return 0;  
}
```

```
gcc -o t foo.o m.o  
m.o:(.bss+0x0): multiple definition of `i'  
foo.o:(.data+0x0): first defined here  
collect2: ld returned 1 exit status
```

Processo de resolução

- O *linker* processa arquivos da esquerda para a direita
 - se é um módulo objeto, adiciona à lista de módulos a serem incluídos no executável, e atualiza lista combinada de símbolos exportados e referências externas
 - se é biblioteca, o *linker* tenta casar as referências externas **computadas até o momento** com símbolos exportados pelos módulos da biblioteca
 - se há pelo menos um casamento para um módulo *m*, adiciona *m* à lista de módulos e atualiza lista de símbolos e referências (módulos da biblioteca também podem ter referências externas!)
- A ordem de especificação das bibliotecas na linha de comando pode alterar o resultado!!!
 - se não houver casamento, módulos não são incluídos
 - biblioteca que define um símbolo deve vir depois de módulos que o referenciem!

Nomes Externos em C

- É importante distinguir / identificar qual declaração é a **definição** (exportação) de um símbolo e quais são **referências** (importações)
 - a definição (deve ser única no programa) estabelece a representação na memória (localização, tipo/tamanho, ...), que é especificada na tabela de símbolos
 - Em C essa distinção não é trivial, e compiladores usam modelos diferentes para identificar definições e referências
- Em ISO C:
 - uma declaração **com inicialização** é uma definição (exportação **forte**)
 - uma declaração que explicita a classe *extern* (sem inicialização) é uma **referência**
 - uma declaração sem *extern* e sem inicialização é uma **tentativa de definição** (exportação **fraca**)
 - poderá ser escolhida como a definição do símbolo se nenhum módulo gerar uma exportação forte

Cuidados e boas práticas

- Uma variável global deve ter um único ponto de definição (em um único arquivo fonte)
 - deve omitir a especificação da classe *extern* e incluir uma inicialização explícita

```
int errcnt = 0;
```

- As declarações que correspondem a referências (importações) devem incluir a especificação da classe *extern*

```
extern int errcnt;
```

- **Um símbolo global deve sempre ser declarado com o mesmo tipo em todos os módulos!**
 - O compilador não tem como garantir, a não ser com o uso de arquivos de *headers* (.h), incluídos por todos os módulos
 - A resolução de referências é feita com base apenas no **nome** do símbolo!
 - a inconsistência de tipos não gera erro de linkedição, e sim um comportamento errado (*bug*) em tempo de execução!

Exemplo de *bug*...

m.c

```
...  
void f(void);  
  
int x = 15213; ←  
int y = 15212;  definição  
  
int main() {  
    f();  
    printf("x = %d y = %d\n", x, y);  
    ...  
}
```

tentativa

```
...  
double x;  
  
void f() {  
    x = -0.0;  
}
```

```
x = 0 y = -2147483648
```

- double ocupa 8 bytes e int ocupa 4 bytes (IA32)
 - a atribuição na função **f** sobrescreverá as posições de memória de **x** e **y**, pois esses símbolos são **definidos** em **m.c**

Dicionário de Relocação

- Informações sobre instruções e dados que referenciam endereços de memória
 - esses endereços não podem ser determinados (ou totalmente determinados) na geração do código objeto
 - funções ou variáveis externas
 - endereço “final” depende da localização da seção de código ou dados correspondente na memória
- Cada entrada possui:
 - a localização (offset) da referência a ser modificada (relocada)
 - o símbolo associado a essa referência
 - tipo de referência
 - endereço absoluto ou relativo ao PC

Relocação de Referências

- Na tabela de símbolos cada entrada (função ou variável) está associada a um valor (offset em relação ao início de sua seção, em seu módulo)
- O *linker* então concatena as seções de código e dados de todos os módulos, atribuindo um *offset* para cada seção de um módulo
 - a tabela de símbolos é atualizada, somando-se ao valor de cada símbolo o *offset* de seu módulo
- As instruções e dados especificados no dicionário de relocação são modificadas com base na tabela de símbolos

Informação de Relocação

m.c

```
int i;
int foo(int j);

int main() {
    int k = foo(i);
    return 0;
}
```

```
00000000 <main>:
 0: 55                push   %ebp
 1: 89 e5             mov    %esp,%ebp
 3: 83 e4 f0          and    $0xffffffff0,%esp
 6: 83 ec 20          sub    $0x20,%esp
 9: a1 00 00 00 00    mov    0x0,%eax
                   a: R_386_32      i
 e: 89 04 24          mov    %eax,(%esp)
11: e8 fc ff ff ff    call   12 <main+0x12>
                   12: R_386_PC32     foo
16: 89 44 24 1c       mov    %eax,0x1c(%esp)
1a: b8 00 00 00 00    mov    $0x0,%eax
1f: c9                leave
20: c3                ret
```

```
gcc -O0 -c m.c
objdump -r -d m.o
```

Carga e Relocação

- Na etapa de amarração as referências à memória são calculadas como *offsets* em relação ao endereço 0 (ou endereço de início “virtual” do programa)
 - na carga do programa para a memória principal, essas referências devem ser novamente relocadas, agora baseadas no endereço “real” do início do programa
- A relocação em tempo de carga/execução pode ser feita:
 - por espaço de endereçamento virtual (tradução de endereços pelo hardware)
 - através de registradores de base de segmento (cálculo de endereço por hardware, também)

Bibliotecas Dinâmicas (compartilhadas)

- Módulos objeto carregados na memória e “ligados” dinamicamente a um programa em tempo de execução
 - *shared objects* (.so) no Unix, *dynamic link libraries* (.dll) no Windows
- Melhor aproveitamento da memória e diminuição do tempo de carga
 - módulos de bibliotecas estáticas são incorporados a **cada** aplicação!
 - módulos de bibliotecas dinâmicas (código) carregados na memória são **compartilhados** por diferentes processos em execução
- A carga e ligação da biblioteca ao programa é realizada por um *dynamic linker*

Ligação com Bibliotecas Dinâmicas

